
# Rapport SR01-Devoir 3

Majuran CHANDRAKUMAR, Xinghua SHAO

Automne 2021

Université de Technologie de Compiègne

# Sommaire

# Introduction

L'objectif de cet exercice est d'implémenter le jeu de vie en utilisant Tkinter pour réaliser l'interface graphique du jeu.

# Définition du jeu

Le jeu de la vie évolue normalement sur un damier infini. Chaque case est occupée par une cellule qui peut être vivante ou morte. A chaque génération, chaque cellule peut naitre, mourir, ou rester dans son état. Les règles qui permettent de passer d'une génération à l'autre sont précises et ont été choisies avec soin pour que l'évolution des organismes soit intéressante et imprévisible.
En premier lieu, notons que sur un damier infini, chaque case a exactement 8 voisins. Pour créer la génération suivante à partir de la génération courante on suit les règles suivantes :

- Une cellule ayant exactement 2 ou 3 voisins vivants survit la génération suivante.
- Une cellule ayant au moins 4 cellules voisines vivantes meurt d'étouffement à la génération suivante.
- Une cellule ayant au plus une cellule voisine vivante meurt d'isolement à la génération suivante.
- Sur une case vide ayant exactement 3 voisins vivants, une cellule naitra à la génération suivante.

Afin d'implémenter ce jeu, on considère le damier infini comme une matrice "torique". Le damier sera représenté par une matrice dont les bords droite et gauche sont reliés entre eux, ainsi que les bords supérieur et inférieur. Par ailleurs, les cellules vivantes seront représentées par des cases rouges et les cellules mortes seront représentées par des cases blanches.

# Fonctions implémentées et interface graphique

## Fonctions pour la modélisation du jeu de vie

- **init_matrice :**

Cette fonction nous permet d'initialiser une matrice de la taille récupérée par le scale « taille de la grille » et d'un pourcentage d'élément de la matrice égale à 1 (ici, une cellule vivante correspond à un élément de la matrice égale à 1 et une cellule morte, à un élément de la matrice égale à 0), également récupérée par le scale « % de vie » de l'interface graphique.
Pour cela, on initiale d'abord la matrice vide. Puis, grâce aux deux boucles for, on parcourt chaque élément de la matrice auquel on associe une valeur aléatoire entre 0 et 1 tout en respectant le pourcentage de vie, entrée en paramètre a l'aide de l'instruction random.choices.

```python
# C'est une fonction permettant d'initialiser une matrice de la taille récupérée par le scale « taille de la grille »
# et d'un pourcentage d'élément de la matrice égale à 1 également récupérée par le scale « % de vie»  de l'interface graphique.
def init_matrice(taille, pourcentage_vie):
    pourcentage_vie = pourcentage_vie / 100 # Convertir la valeur de pourcentage_vie en une valeur entre 0 et 1
    values = [0, 1] # C'est une liste de valeur propable pour initialiser une matrice
    weights = [1 - pourcentage_vie, pourcentage_vie] # C'est la probabilité de 0 et 1, 1-pourcentage_vie est la probabilité pour 0 et pourcentage_vie est la probabilité pour 1
    matrice = numpy.zeros((taille, taille)) # initialiser une matrice vide
    # On utilise la fonction choices de la module "random" pour obetenir 0 ou 1 par la probabilité conrrespondante
    for i in range(0, taille):
        for j in range(0, taille):
            matrice[i][j] = choices(values, weights, k=1)[0] # Ce que la fonction retourne, c'est une liste, donc on prend le premier élément de la liste
    return matrice
```

Code de la fonction init_matrice

- **voisin :**

Cette fonction nous permet pour un élément d'une matrice de compter son nombre de voisins vivants c'est-à-dire son nombre de voisins égale à 1.

```python
# C'est une fonction permettant de calculer le nombre de voisin autour d'un élément dans la matrice
def voisin(matrice, nligne, ncolonne):
    nb_voisin = 0
    for i in range(nligne - 1, nligne + 2):
        if (matrice[(i + len(matrice)) % len(matrice)][(ncolonne - 1 + len(matrice)) % len(matrice)] != 0):
            nb_voisin += 1
        if (matrice[(i + len(matrice)) % len(matrice)][(ncolonne + 1 + len(matrice)) % len(matrice)] != 0 ):
            nb_voisin += 1
        if (i + len(matrice)) % len(matrice) != nligne:
            if matrice[(i + len(matrice)) % len(matrice)][ncolonne] != 0:
                nb_voisin += 1
    return nb_voisin
```

Code de la fonction voisin

- **mise_a_jour :**

Cette fonction nous permet de créer la génération suivante à partir de la génération courante en suivant les règles décrites au-dessus.

Pour cela, à l'aide d'une boucle for, pour chaque élément de la matrice, on calcule son nombre de voisins vivants à l'aide de la fonction « voisin ». Ensuite, en fonction du nombre de voisins vivants déterminés, on actualise ou non l'élément de la matrice. Ainsi,

- Si le nombre de voisins vivants est égale à 2, l'élément de la matrice reste inchangé
- Si le nombre de voisins vivants est égale à 3, l'élément de la matrice passe à 1.
- Sinon, l'élément de la matrice passe à 0.

A la fin du parcours de la matrice entrée en paramètre, on retourne la nouvelle matrice.

```python
# C'est une fonction permettant de générer une nouvelle génération suivante à partir de la génération courante
def mise_a_jour(matrice):
    nb_voisin = 0
    nouvelle_matrice = numpy.zeros((len(matrice), len(matrice))) # initialiser une nouvelle matrice vide en utilisant la fonction numpy.zeros()
    # On parcourt toutes les cases dans la matrice
    for i in range(0, len(matrice)):
        for j in range(0, len(matrice)):
            nb_voisin = voisin(matrice, i, j) # Obtenir le nombre de voisins survivants de l'élément actuelle
            if nb_voisin == 2:
                nouvelle_matrice[i][j] = matrice[i][j]
            elif nb_voisin == 3:
                nouvelle_matrice[i][j] = 1
            else:
                nouvelle_matrice[i][j] = 0
    return nouvelle_matrice # retourner la nouvelle matrice
```

Code de la fonction mise a jour

# Interface graphique

- Création de la fenêtre avec titre et taille :

Tout d'abord, pour la création de la fenêtre de l'interface graphique, il suffit d'utiliser la commande tk. Ensuite, pour le titre de la fenêtre, on a recours à la commande title et enfin pour la taille, on opte pour la commande geometry.

```
# L'interface graphique
window = Tk()
window.title("SR01 Jeu de la vie")
window.geometry("800x600")
```

Code  pour création de la fenêtre

- Création des deux frames :

Pour la création du frame contenant les boutons, on utilise la commande Frame qui prendra comme width 200, height 600 et de background gris (#f0eeec). Ensuite, à l'aide de la commande pack, on place le frame des boutons dans la partie droite de la fenêtre.
Pour la création du frame contenant la grille, on utilise la commande Frame qui prendra comme width 600, height 600 et de background blanc (#ffffff). Ensuite, à l'aide de la commande pack, on place le frame de la grille dans la partie gauche de la fenêtre.

```
# Frame de la grille
Frame2 = Frame(window, width=600, height=600, bg="#ffffff")
Frame2.pack(side=LEFT)

# Frame des boutons
Frame1 = Frame(window, width=200, height=600, bg="#f0eeec")
Frame1.pack_propagate(FALSE)
Frame1.pack(side=RIGHT)
```

Code pour création des deux frames

- Création des boutons :

Pour la création des boutons, il suffit d'utiliser la commande Button où on indique le frame où se situe le bouton, le nom du bouton, la couleur du bouton et de la police d'écriture et enfin la fonction qui s'exécute lorsque le bouton est activé par un clic gauche. Ensuite, à l'aide de la commande pack, on place le bouton en haut ou en bas du frame et on indique que le bouton remplit toute la ligne avec fill=X.

```
# Buttons dans Frame1
Button1 = Button(Frame1, text="Quitter", bg="#d5d3d2", fg="#1d4699", command=quitter).pack(side=BOTTOM, fill=X)

Button2 = Button(Frame1, text="Lancer", bg="#d5d3d2", fg="#1d4699", command=lancer).pack(side=TOP, fill=X)

Button3 = Button(Frame1, text="Arreter", bg="#d5d3d2", fg="#1d4699", command=arreter).pack(side=TOP, fill=X)

Button4 = Button(Frame1, text="Initialiser", bg="#d5d3d2", fg="#1d4699", command=initialiser).pack(side=TOP, fill=X)
```

Code pour création des boutons

- Création des scales :

Pour la création des scales, il suffit d'utiliser la commande Scale où on indique le frame où se situe le scale, le nom du scale, la couleur de la police d'écriture, l'orientation du scale (ici horizontale), la variable affectée au scale et enfin la borne des valeurs que peut prend la variable. Ensuite, à l'aide de la commande pack, on place le scale en bas du frame et avec la commande set, on indique la valeur initiale que prend la variable du scale.

```
# variable pour récupérer la vitesse dans le scale 'Vitesse'
vitesse = IntVar()
Scale1 = Scale(Frame1, variable=vitesse, from_=1, to=15, orient=HORIZONTAL,  label = 'Vitesse', fg='#1d4699')
Scale1.pack(side=BOTTOM)
Scale1.set(1) # la vitesse par défaut est 1

# variable pour récupérer le pourcentage_vie dans le scale '% de vie'
pourcentage_vie = IntVar()
Scale2 = Scale( Frame1, variable=pourcentage_vie ,from_=10, to=90, orient = HORIZONTAL, label = '% de vie', fg='#1d4699' )
Scale2.pack(side=BOTTOM)
Scale2.set(20) # le pourcentage_vie par défaut est 20

# variable pour récupérer la taille dans le scale 'Taille de la grille'
taille = IntVar()
Scale3 = Scale( Frame1, variable=taille ,from_=10, to=100, resolution=5, orient = HORIZONTAL,  label = 'Taille de la grille', fg='#1d4699' )
Scale3.pack(side=BOTTOM)
Scale3.set(30) # la taille par défaut est 30
```

Code pour création des scales

# Fonctions pour l'interface graphique

- **draw_grid :**

Cette fonction permet de dessiner une grille vide. Elle prend comme paramètre le width du frame dédié à la grille et la taille récupérée par le scale « taille de grille ».

Ainsi, pour cela, dans un premier temps, nous allons détruire tous les widgets à l'intérieur du frame à l'aide de destroy en ciblant les widgets enfants du frame dédiée à la grille avec winfo_children. Ensuite, on détermine la taille de chaque case de la grille. Ensuite, on crée un canvas qui prend tout le frame dédié à la grille avec un background blanc. A l'aide de la boucle for, on dessine une ligne horizontale et verticale de couleur noir sur le canvas créé avec l'instruction create_line tant qu'on n'a pas dépassé la taille entrée en paramètre.

```
# C'est une fonction permettant de dessiner des grilles
def draw_grid(width, taille):
    for widget in Frame2.winfo_children(): # Avant de dessiner des nouvelles grilles, on détruit des grilles existantes dans Frame2
        widget.destroy()
    grid = Canvas(Frame2, width=width, height=width, background="white") # Création d'un objet de type Canvas dans le Frame2
    sizePerCellule = width / taille # la longeur de chaque cellules
    x, y = 0, 0
    for i in range(taille):
        x += sizePerCellule
        y += sizePerCellule
        grid.create_line(x,0,x,width, fill="black") # lignes verticales
        grid.create_line(0,y,width,y, fill="black") # lignes horizontales
    grid.pack()
    return grid
```

Code de la fonction draw grid

- **fill_grid :**

Cette fonction permet de dessiner une grille en différenciant les cellules vivantes et mortes. Elle prend une matrice et une grille.

Ainsi, pour cela, dans un premier temps, nous allons détruire les éléments rectangles à l'intérieur de la grille entrée en paramètre à l'aide de delete. Ensuite, on détermine la taille de chaque case de la grille. Ensuite, grâce aux deux boucles for, on parcourt chaque élément de la matrice entrée en paramètre. Si un élément de la matrice est égal à 1 alors la case de la grille correspondante a cet élément sera de couleur rouge a l'aide de l'instruction create_rectangle.

```
# C'est une fonction permettant de dessiner une grille en différenciant les cellules vivantes et mortes
# les cellules vivantes: rouge
# les cellules mortes: blanc
def fill_grid(matrice, grid):
    grid.delete("rectangles") # Avant de mettre les cellules vivantes en rouge, on efface les cellules précédantes
    sizePerCellule = 600 / len(matrice) # la longeur de chaque cellules
    for i in range(len(matrice)):
        for j in range(len(matrice)):
            if matrice[i][j] == 1: # Si la valeur d'une cellule = 1, on dessine un carré rouge dans cette cellule
                grid.create_rectangle(j*sizePerCellule, i*sizePerCellule, (j+1)*sizePerCellule, (i+1)*sizePerCellule, fill="red", tag="rectangles")
```

Code de la fonction fill grid

- **initialiser :**

Cette fonction nous permet d'initialiser une grille et de l'afficher dans l'interface

Ainsi, pour cela, dans un premier temps, on initialise une matrice à l'aide de la fonction « init_matrice ». Ensuite, on dessine une grille vide avec la fonction « draw_grid » puis on remplit la grille avec la fonction « fill_grid » avec comme paramètre la matrice initialisée et la grille vide.

```
# C'est une fonction permettant d'initialiser une grille et de l'afficher dans l'interface
def initialiser():
    arreter() # arrêter le jeu avant d'initialiser
    global matrice
    matrice = init_matrice(taille.get(), pourcentage_vie.get()) # Générer une matrice d'initialisation
    global grid
    grid = draw_grid(600, taille.get()) # Dessiner la grille
    fill_grid(matrice, grid) # remplissage de la grille avec la matrice d'initialisation
```

Code de la fonction initialiser

- **lancer :**

Cette fonction nous permet de lancer le déroulement du jeu de la vie.

Ainsi, pour cela, on affecte la valeur FALSE a la variable globale STOP. Ensuite, on exécute la fonction « run ».

```
# C'est une fonction permettant de lancer le déroulement du jeu de la vie
def lancer():
    global stop
    stop = FALSE
    run()
```

Code de la fonction lancer

- **run :**

Cette fonction nous permet de continuer le déroulement du jeu de la vie.
Ainsi, pour cela, tant que le valeur globale STOP prend la valeur FALSE, on détermine la matrice de la génération suivante de celle actuelle a l'aide de la fonction « mise_a_jour» qui prend comme paramètre la matrice de la génération actuelle. Ensuite, on remplit la grille avec cette nouvelle matrice en utilisant la fonction « fill_grid». Ensuite, on détermine la valeur de délai entre l'affichage de deux générations. Une fois le délai passé, la fonction « run » se réexécute avec after.

```
# C'est une fonction permettant de continuer le déroulement du jeu de la vie
def run():
    global stop
    if stop == FALSE: # Si stop = FALSE, on continue le déroulement du jeu de la vie
        global matrice
        new_matrice = mise_a_jour(matrice) # mise a jour de la matrice
        fill_grid(new_matrice, grid) # remplissage de la grille avec la matrice
        matrice = new_matrice
        delay = int(1 * 1000 / vitesse.get()) # C'est le temps entre deux affichages, plus la vitesse est grande, plus le temps entre deux affichages est court
        window.after(delay, run) # lancement de la fonction run après delay s
```

Code de la fonction run

- **arreter :**

Cette fonction nous permet d'arrêter le déroulement du jeu de la vie.
Ainsi, pour cela, on affecte la valeur TRUE a la variable globale STOP. Ainsi, la fonction « run » ne sera pas exécutée une nouvelle fois car il faut que la variable STOP prenne la valeur FALSE.

```
# C'est une fonction permettant d'arrêter le déroulement du jeu de la vie
def arreter():
    global stop
    stop = TRUE
```

Code de la fonction arreter

- **quitter :**

Cette fonction nous permet de quitter l'interface graphique et de stopper le programme.
Ainsi, pour cela, il suffit de détruire la fenêtre créée à l'aide de destroy.

```
# C'est une fonction permettant de de stopper le programme et quitter l'interface graphique.
def quitter():
    window.destroy()
```

Code de la fonction quitter

# Résultat de l'interface graphique